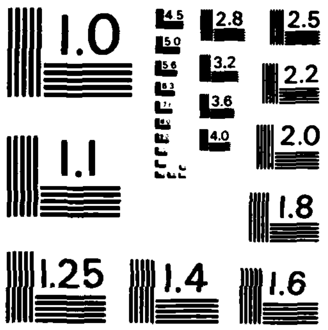
AD-A161 434 COMPUTING MINIMUM SPANNING TREES ON A FAT-TREE ARCHITECTURE(U) MASSACHUSETTS INST OF TECH CAMBRIDGE DEPT OF ELECTRICAL ENGIN.. B M MAGGS OCT 85 1/1

UNCLASSIFIED N00014-80-C-0622 F/G 12/1 NL

END
FILMED
DTIC


1.0
1.1
1.25
1.4
1.6
1.8
2.0
2.2
2.5
2.8
3.2
3.6
4.0
4.5
5.0
5.6
6.3

DEPARTMENT OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE
MASSACHUSETTS INSTITUTE OF TECHNOLOGY
CAMBRIDGE, MASSACHUSETTS 02139

AD-A161 434

VLSI Memo No. 85-269 October 1985

Computing Minimum Spanning Trees On a Fat-Tree Architecture*

Bruce M. Maggs**

ABSTRACT

[illegible] parallel [illegible] sq log beta

This paper presents two algorithms for computing the minimum spanning forest of an input graph on a fat-tree architecture. One algorithm is deterministic, and the other probabilistic. The deterministic algorithm generates $O(\cancel{\log}^3|V|)$ message sets, each of which can be delivered in $O(\beta(G))$ delivery cycles. The probabilistic algorithm generates $O(\cancel{\log}^2|V|)$ message sets, each of which can be delivered in $O(\beta(G))$ delivery cycles.

*Submitted to the Department of Electrical Engineering and Computer Science, MIT, in partial fulfillment of the requirements for the degree of Bachelor of Science, May, 1985. This research was supported in part by the Defense Advanced Research Projects Agency under contract number N00014-80-C-0622.

Computing Minimum Spanning Trees on a Fat-Tree Architecture

by

Bruce MacDowell Maggs

Submitted to the Department of
Electrical Engineering and Computer Science
in Partial Fulfillment of the
Requirements of the Degree of

Bachelor of Science

at the

Massachusetts Institute of Technology

May, 1985

Signature of Author______________________________
Department of Electrical Engineering and Computer Science
May 17, 1985

Certified By______________________________
Professor Charles E. Leiserson, Thesis Supervisor

Accepted By______________________________
Professor David Adler, Chairman, Department Committee

COMPUTING MINIUMUM SPANNING TREES

On a Fat-Tree Architecture [1]

Bruce M. Maggs
Laboratory for Computer Science
Massachusetts Institute of Technology
Cambridge, Massachusetts 02139

Abstract–

This paper presents two algorithms for computing the minimum spanning forest of an input graph on a fat-tree architecture. One algorithm is deterministic, and the other probabilistic. The deterministic algorithm generates $O(\log^3 |V|)$ messages sets, each of which can be delivered in $O(\beta(G))$ delivery cycles. The probabilistic algorithm generates $O(\log^2 |V|)$ message sets, each of which can be delivered in $O(\beta(G))$ delivery cycles.

[1]This thesis describes joint work with Professor Charles E. Leiserson. This research was supported in part by the Defense Advanced Research Projects Agency under Contract $N00014-80-C-0622$.

1 Introduction

In this paper we present a parallel algorithm for computing the minimum spanning forest of a graph on a fat-tree architecture. That is, given graph $G = (V, E)$ where the edges in E are weighted, we want to find a set of edges forming a minimum spanning tree for each connected component. We will analyze the running time not only in terms of $|V|$ and $|E|$, but also in terms of how efficiently the graph has been embedded in the fat-tree.

Parallel algorithms for computing the connected components or the minimum spanning forest of an input graph have been presented for numerous parallel architectures [AS, B, H, HCS, KR, SV]. Awerbuch and Shiloach, for example, have presented a minimum spanning forest algorithm with running time $O(\log|V|)$ [AS]. Their algorithm is intended for a PRAM (Parallel Random Access Memory) model with CRCW (Concurrent Read and Concurrent Write) capabilities. Each of the $|E| + |V|$ processors in this model has access to every word of a shared memory. While this model is very powerful, the connectivity required to build such a shared memory is so high that it may be impractical except on a small scale. Other authors have presented minimum spanning tree algorithms for less highly connected but also less general architectures. In particular, Bentley has presented a minimum spanning tree algorithm for a specialized tree architecture containing $|V|$ processors [B]. Bentley's algorithm has running time $O(|V| \log |V|)$. In this paper we present a minimum spanning forest algorithm for a new class of universal routing networks introduced in [L] called fat-trees.

Leiserson has shown that under the assumption that only $O(A)$ bits may enter or leave a region R with surface area A in unit time, fat-trees have the following universality property: given any routing network R consisting of some fixed amount of hardware (a set P of processing elements wired together in volume V), there exists a fat-tree built with the same amount of hardware that can simulate the original network at a cost of a factor of $O(\log^3 |P|)$ in time. Thus for a given amount of hardware, a fat-tree can in theory be used solve a problem, such as computing the minimum spanning forest of a graph, in almost optimal time. Leiserson's theorem indicates that fat-trees are a powerful class of routing networks. His paper, however, explains only how to simulate other routing networks and says nothing about how to design efficient algorithms specifically for the fat-tree. In this paper we describe new data structures and techniques that may be useful in future fat-tree algorithms.

We also introduce a new parameter to the running times of parallel algorithms. The running times of sequential and parallel algorithms are typically parameterized by the size of the input. For example, the two parallel algorithms mentioned above have running times parameterized by the number of vertices and edges in the input graph G. The new parameter, which we will call the base load factor of G, $\beta(G)$, is a measure of the communication congestion that occurs when some primitive operation is performed in parallel on the input data. We will embed each vertex $v \in G$ in a different processor and our primitive operation will be for each vertex to simultaneously pass a message to each neighboring vertex. In this algorithm, the communication congestion of every message set, and consequently of the entire algorithm can be expressed in terms of $\beta(G)$.

The remainder of this paper is organized in the following manner. In section 2 we define a fat-tree architecture and the concepts of message sets and their load factors. In the sections 3 and 4 we describe the message set routing results of Leiserson and Greenberg [LG] and prove a short lemma extending these results. In section 5 we describe the parallel minimum spanning forest algorithm that we are going to implement. Our implementation requires the auxilliary data structures and subalgorithms that are described in section 6. Following these descriptions, we present our minimum spanning forest algorithm in section 7. In sections 8 and 9 we present three more subalgorithms of the minimum spanning forest algorithm. Section 10 is an analysis of the running time of the algorithm. A message set synchronization scheme using the ideas of this paper is described in section 11. We conclude with a few comments on future fat-tree research.

2 Fat-Trees

A fat-tree is depicted in Figure 1. The underlying structure of a fat-tree is a complete binary tree. The leaves of the binary tree are processor elements, the internal nodes are switches, and the edges are communication channels. In general, the capacities of the communication channels increase as the tree is traversed from the leaves to the root. More formally, a fat-tree is an ordered triple $FT = (P, N, C)$ where P is the set of processors found at the leaves, N is the set of switches found at the internal nodes, and C is the set of channels found at the edges. We let $\text{cap}(c)$ denote the capacity of a channel $c \in C$, that is, the number of messages that may be simultaneously sent through c. In the fat-trees that we will consider, the channels are unidirectional and paired. That is, for each channel going up the fat-tree there is a corresponding channel with the same capacity going down the fat-tree. Each processor p has a unique address in the fat-tree, $l(p)$. In Figure 1, for example, $l(p_1)$ is 010. We assume that each processor has a copy of its own address.

Definition 1 *A message set $M \subseteq P \times P$ is a set of messages where $(p_1, p_2) \in M$ is a message from processor p_1 to processor p_2.*

Because the underlying structure of a fat-tree is a tree, message (p_1, p_2) must traverse the unique path from p_1 to p_2 in FT.

Definition 2 *Let* $\text{load}(M, c)$ *be the number of messages in message set M that must traverse channel $c \in C$.*

Definition 3 *M is called a one-cycle message set if for all $c \in C$,*

$$\text{load}(M, c) \leq \text{cap}(c).$$

Because none of the channel capacities are exceeded, all of the messages in a one-cycle message set can be delivered in one message delivery cycle.

Definition 4 *The load factor, $\lambda(M, c)$, of a channel $c \in C$ due to a message set M is*

$$\lambda(M, c) = \frac{\text{load}(M,c)}{\text{cap}(c)}.$$

Definition 5 *The load factor of FT due to M, $\lambda(M)$, is*

$$\lambda(M) = \max_{c \in C} \lambda(M, c).$$

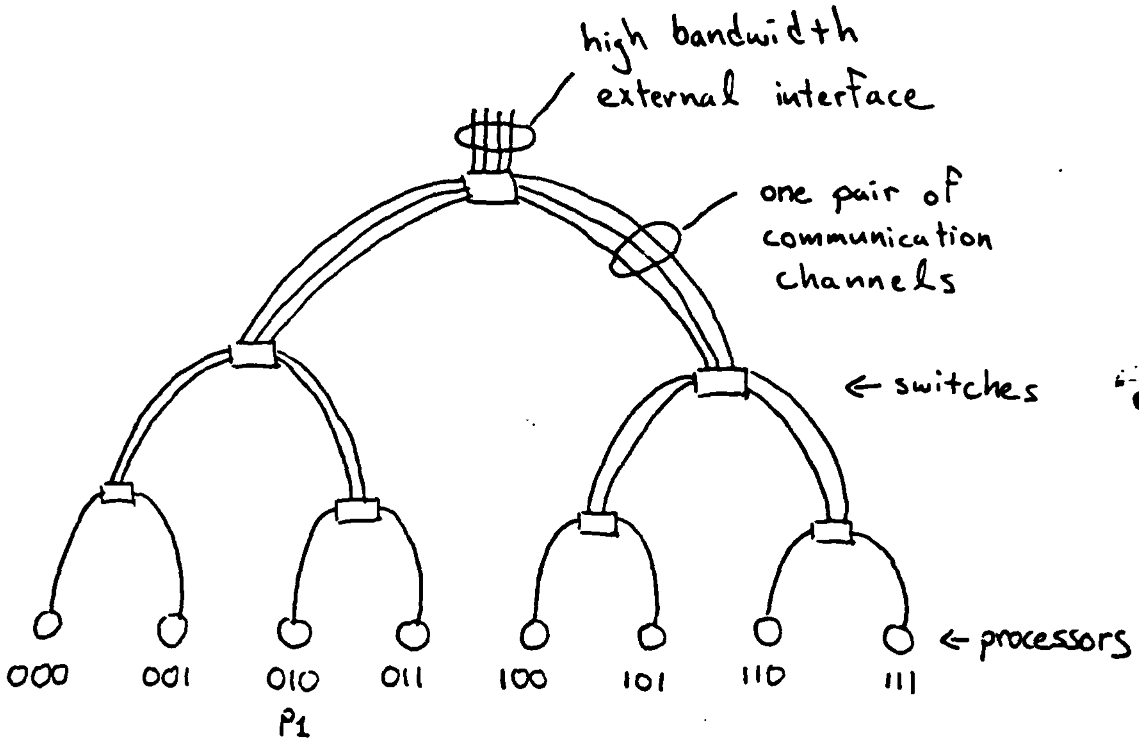

high bandwidth external interface
one pair of communication channels
← switches
← processors
000
001
010
011
100
101
110
111
P1

Figure 1: A Fat-tree Architecture

3 Routing Message Sets

Figure 2 shows an input graph G embedded in a fat-tree FT. In this figure, the channel capacities of FT and the edge weights of G have been omitted for clarity. Each vertex $v \in G$ is assigned to its own processor, $\phi(v)$. Where the context removes any ambiguity we will, for simplicity, let v denote $\phi(v)$ and the address of $\phi(v)$, $l(\phi(v)$. Let v have neighbors $v_1, v_2, ..., v_k$ in G. In processor $\phi(v)$ we store the adjacency list of v, $(v_1, w_1), (v_2, w_2), ..., (v_k, w_k)$, where w_i denotes the weight of the edge connecting v and v_i. In this example, v_1 has been embedded in processor p_1. In p_1 we store the adjacency list (010, 1),(100, 2), (110, 2) (111, 2).

Let G' be the graph that results when each edge in G is replaced by two oppositely directed edges. In general we will use the symbol $'$ to denote the operation of replacing each edge of an undirected graph with two oppositely directed edges. Let $M_{G'}$ be the message set that arises when each vertex in G' sends a message to each of its neighbors. We will use this primitive operation to determine the base load factor of the input graph G. Leiserson and Greenberg have shown [LG] that an arbitrary message set M can be broken up into one-cycle message sets on-line and can, with high probability, be routed in $O(\lambda(M) + \log|P| \log\log|P|)$ delivery cycles. This on-line routine algorithm assumes the existence of a hardware mechanism to synchronize the sending of messages by the processors in P. In a later section we will show how message set synchronization can be accomplished with no dedicated hardware other than increased channel capacities. Using either synchronization scheme, a message set M will still, with high probability, be delivered in $O(\lambda(M) + \log|P| \log\log|P|)$ delivery cycles.

The choice of a particular on-line message set routing algorithm is not important to the understanding of the minimum spanning tree algorithm. Throughout the paper we will assume that we have some mechanism for synchronizing message sets and delivery cycles within those message sets, and for deciding which messages belong in which delivery cycles. We assume that each processor knows when the routing of a message set M begins and ends, and when each delivery cycle used to route M has begins and ends. If we use the algorithm of Leiserson, and Greenberg, we can send $M_{G'}$ in $O(\lambda(M_{G'}) + \log|P| \log\log|P|)$ delivery cycles. We define $\beta(G)$, the base load factor of G, to be $\lambda(M_G) \div \log|P| \log\log|P|$. We will analyze the number of delivery cycles used by the algorithm in terms of $\beta(G)$ and $|V|$.

In the literature, graph algorithms in which vertices are allowed to communicate only with their neighbors are called "distributed". In such algorithms a message from one vertex to another may have to pass through $O(|V|)$ intermediate vertices. Although we embed each vertex in its own processor, our MSF algorithm is not in this sense distributed. We may pass a message from v_1 to v_2 even when they are not neighbors in G.

4 The Shortcut Lemma

Figure 3 shows a message set M_0 in which processor p_1 sends a message to p_2 and p_2 sends a message to p_3. The following lemma shows that we may replace these two messages in M_0 with a message directly from p_1 to p_3 without increasing the load factor of M_0.

Lemma 1 *The Shortcut Lemma*

Let p_1, p_2, and p_3, be leaves of a fat tree. Suppose p_1 is sending a message to p_2 and p_2 is sending a message to p_3 in message set M_0. Then the load factor of the message set that results when these two messages are replaced by a message directly from p_1 to p_3,

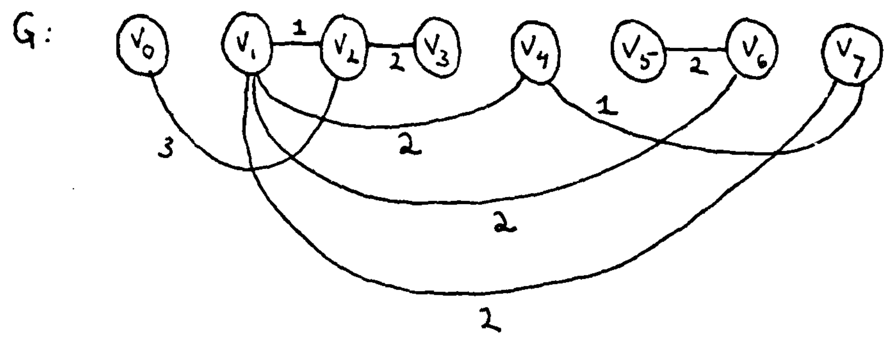
$$M = (M_0 \cup \{(p_1,p_3)\}) - \{(p_1,p_2),(p_2,p_3)\},$$

is not greater than the load factor of the original message set M_0. That is,

$$\lambda(M) \le \lambda(M_0).$$

Proof: It will suffice to show that $\text{load}(M,c) \le \text{load}(M_0,c)$ for each $c \in C$, since by definition $\lambda(M_0,c) = \frac{\text{load}(M_0,c)}{\text{cap}(c)}$. Since the underlying structure of FT is a tree, and a tree cannot contain any simple cycles, the paths $p_1 \twoheadrightarrow p_2$ and $p_2 \twoheadrightarrow p_3$ must contain the unique simple path $p_1 \twoheadrightarrow p_3$. Therefore message (p_1,p_3) passes through a channel c only if either (p_1,p_2) or (p_2,p_3) does also. $\Box$

The Shortcut Lemma can be extended to show that we can replace any subset of M_0 that forms a path of messages from p_1 to p_N with a single message directly from p_1 to p_N.



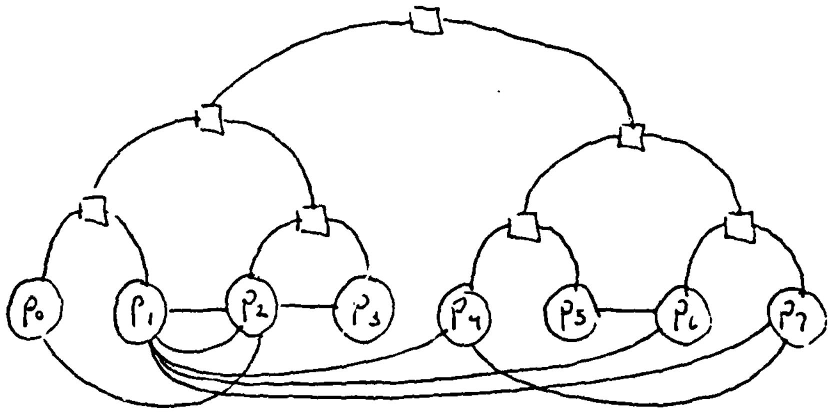


Figure 2: Embedding a Graph in a Fat-tree 2

Corollary 1 *Extended Shortcut Lemma*

Let $(p_1,p_2),(p_2,p_3),...,(p_{N-1},p_N) \in M_0$. Suppose we replace messages (p_1,p_2) through (p_{N-1},p_N) with a single message (p_1,p_N) in message set M. That is, let $M = (M_0 \cup \{(p_1,p_N)\}) - \{(p_1,p_2),(p_2,p_3),...,(p_{N-1},p_N)\}$. Then the load factor of M will not be greater than the load factor for M_0.

$$\lambda(M) \leq \lambda(M_0)$$

Proof: The proof is by induction on N. □

In general, we will pass a message from v_1 to v_2 in the minimum spanning forest algorithm only if there is a path in G' from v_1 to v_2. Furthermore, in any set of messages (v_i,v_j) that we send, there is some set of paths from the v_i to the v_j such that no edge in E' is traversed more than once. In Figure 4 this paradigm has been violated. We can remove message (p_1,p_2) to shortcut $(p_1,p_2),(p_2,p_3)$, but cannot remove it again to shortcut $(p_1,p_2),(p_2,p_4)$. However, if we follow this paradigm then by the Extended Shortcut Lemma the load factor of every message set will be less than or equal to $\lambda(M_{G'})$ and therefore will, with high probability be delivered in $\beta(G)$ or fewer delivery cycles.

5 Sollin's Algorithm

Our minimum spanning forest algorithm is an implementation of the following parallel algorithm attributed to Sollin in [GH]. We want to find a set of edges, E_T, that forms a minimum spanning tree for each connected component of G. At each stage of the algorithm, let $T_1,T_2,...,T_l$ denote the subtrees formed by the edges in E_T.

Algorithm 1 *Sollin's Algorithm*

```
Each vertex $v_i$ is an isolated subtree $T_i$ of $G$.

WHILE there are edges not in $E_T$ connecting the $T_i$ DO
```

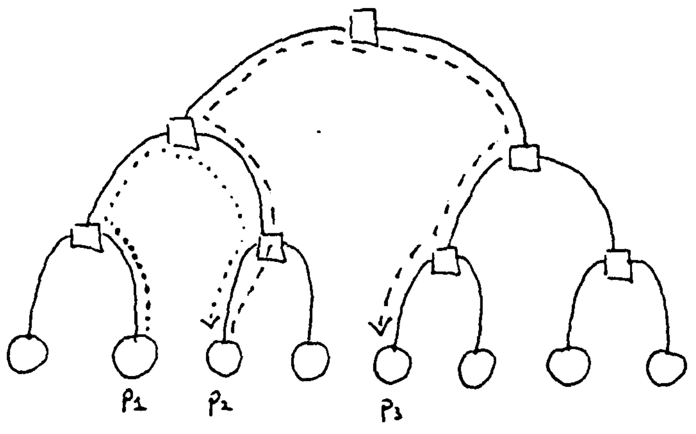


Figure 3: Shortcutting Messages

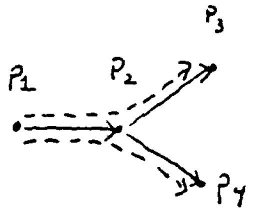


Figure 4: Illegal Shortcutting

Simultaneously select, for each subtree T_i, the edge of smallest weight connecting a vertex $u \in T_i$ with a vertex $v \in T_j, i \neq j$. If there is more than one edge with the same smallest weight, break ties by giving each edge $u \overset{e}{—} v$ a label $k(e) = (\max(u,v), \min(u,v))$ and choosing the edge with the smallest label. Add the selected edge to E_T.

In each iteration of Sollin's algorithm we want to quickly gather information about all of the edges adjacent to a subtree. In the following section we explain how to build a "communication tree" for each subtree T_i through which we can quickly gather this information. The technique we use is reminiscent of the Euler Tour Technique introduced by Tarjan and Vishkin in [TV]. We will compute the Euler tour of each subtree and from this tour build a communication efficient communication tree.

6 Communication Trees

6.1 Euler Cycles of Trees

Let $T_i = (V_i, E_i)$ be a tree where $V_i \subseteq V$ and $E_i \subseteq E$. Let $T_i' = (V_i, E_i')$ be the directed graph that results when each edge in E_i is replaced with 2 oppositely directed edges in E_i'. Clearly T_i' is connected and the in-degree and out-degree of each vertex in T_i' are equal. An elementary result of graph theory is that in any directed graph where for each vertex the in-degree and out-degree are equal there exists a directed Euler cycle [E]. Let C_i' be a directed Euler cycle of T_i'.

A typical step in our minimum spanning forest algorithm is to merge the directed Euler cycles of two vertex disjoint trees $T_u = (V_u, E_u)$ and $T_v = (V_v, E_v)$ connected by an edge e between vertices $u_i \in V_u$ and $v_j \in V_v$ to form the larger directed Euler cycle of $T_{uv} = (V_u \cup V_v, E_u \cup E_v \cup \{e\})$ Let $u_i \xrightarrow{e_{ij}} v_j$ and $v_j \xrightarrow{e_{ji}} u_i$ be two oppositely directed edges between the endpoints of e and let C_u' and C_v' be directed Euler cycles of T_u' and T_v'.

$$\begin{aligned} C_u' &= u_0 \rightarrow u_1 \rightarrow u_2 \rightarrow \ldots \rightarrow u_{i-1} \rightarrow u_i \rightarrow u_{i+1} \rightarrow \ldots \rightarrow u_0 \\ C_v' &= v_0 \rightarrow v_1 \rightarrow v_2 \rightarrow \ldots \rightarrow v_{j-1} \rightarrow v_j \rightarrow v_{j+1} \rightarrow \ldots \rightarrow v_0 \end{aligned}$$

Figure 5 illustrates the process of merging C'_u and C'_v to form the larger directed Euler cycle of the tree $T'_{uv} = (V_u \cup V_v, E'_u \cup E'_v \cup \{e_{ij}, e_{ji}\})$ Cycles C'_u and C'_v are broken apart at vertices u_i and v_j, and then merged by connecting u_i and v_j with edges e_{ij} and e_{ji}. The resulting cycle,

$$\begin{aligned} C'_{uv} &= u_0 \rightarrow \ldots \rightarrow u_{i-1} \rightarrow u_i \rightarrow v_j e_{ij} \rightarrow v_{j+1} \rightarrow \ldots \\ &\rightarrow v_0 \rightarrow \ldots \rightarrow v_{j-1} \rightarrow v_j \rightarrow u_i e_{ij} \rightarrow u_{i+1} \rightarrow \ldots \rightarrow u_0 \end{aligned}$$

is a directed Euler cycle of T_{uv}.

Throughout the execution of the algorithm we will maintain a set of subtrees of G, $\{T_1, T_2, ..., T_l\}$, and a set of directed Euler cycles of those trees $\{C'_1, C'_2, ..., C'_l\}$. Each vertex $v \in V$ will belong to exactly one tree T_i and to the Euler cycle C'_i of T'_i. We will repeatedly merge the T_i and their directed Euler cycles, the C'_i by adding connecting edges. When the algorithm terminates, the condition $u, v \in V_i \iff u, v \in C'_i \iff u, v$ in same connected component of G will be satisfied.

6.2 Building Communication Trees

In describing the following communication tree construction algorithm, we will consider only one of the subtrees of G, T_i. The algorithm, however, runs simultaneously on all of the subtrees formed by the edges in E_T with no communication or interference between them.

The communication tree construction algorithm requires a subroutine that pairs vertices in a cycle. In a later section we will describe two algorithms that perform this pairing, one deterministic, and the other probabilistic.

We build a communication tree from the bottom up by repeatedly merging subtrees of the communication tree to form larger subtrees. We keep the roots of the subtrees in a cycle in the order that they appear in the cycle C' and pick pairs of them to merge. When the mergers are complete we construct a new cycle of subtree roots by removing the vertices that are no longer roots.

In Figure 6 we show the how a pair of subtrees are merged. When we merge two subtrees we want the inorder traversal of the resulting subtree to be the same as the concatenation of the inorder traversals of the merging

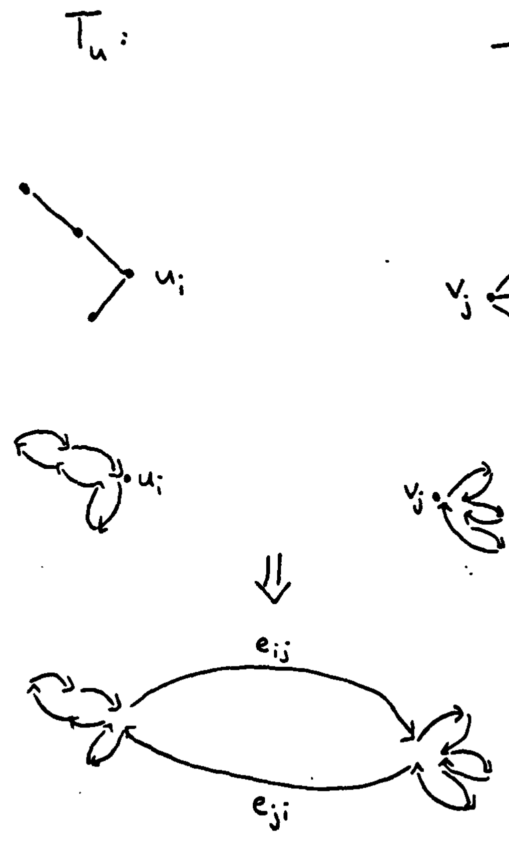


Figure 5: Merging Directed Euler Cycles

subtrees. We do this by making the rightmost leaf of the left subtree the root of the resulting subtree. The left and right roots of the merging subtrees then become the left and right children of the new root. In the inorder traversal of the final communication tree, the nodes appear in the same order that they appear in C'.

In Figure 7 we show a cycle of subtree roots before the mergers have taken place. The circled pairs of vertices are the roots of the subtrees that will merge. In Figure 8 we show the cycle of subtree roots after the mergers have occurred. In each merge, the rightmost leaf of the left subtree has become the new root.

Algorithm 2 *Communication Tree Construction Algorithm*

```
Let $R^i$ denote a cycle of the roots of the subtrees of communication
tree $CT$ after $i$ iterations of the communication tree construction
algorithm. Let $T_j^i$ denote a subtree of $CT$ with root $r_j^i$ and rightmost
leaf $l_j^i$ after $i$ iterations of the algorithm.

Initially, $i = 0$ and $R^0 = C'$.

WHILE $R^i$ contains more than one root DO
    IF $R^i$ contains only two subtree roots, $r_j^i$ and $r_k^i$
        THEN
            Let the vertex with the smaller label, $r_j^i$ or $r_k^i$ be consid-
            ered the vertex on the left.
        ELSE
            Use either the deterministic or the probabilistic pairing
            algorithm to pair off roots in $R^i$.
    For each ordered pair of roots $(r_j^i, r_k^i)$
        1. Let $T_{jk}^{i+1}$ be the union of $T_j^i$ and $T_k^i$.
        2. Remove from $T_{jk}^{i+1}$ the edge connecting $l_j^i$ to his father.
            (a) $r_j^i$ sends $l_j^i$ a message indicating that $l_j^i$ is to become a
                subtree root.
```

   (b) l_j^i sends his father a message indicating that he is no longer a leaf
3. Make r_j^i the left child of l_j^i.
4. Make r_k^i the right child of l_j^i.
   (a) r_j^i sends r_k^i a message containing the identity of l_j^i
   (b) r_k^i sends l_j^i a message containing the identity of r_k^i
5. Let $r_{jk}^{i+1} = l_j^i$.
6. Let $l_{jk}^{i+1} = l_k^i$.
   (a) r_k^i sends l_j^i a message containing the identity of l_k^i
7. Let $R^{i+1} = R^i$.
8. Replace r_j^i and r_k^i in R^{i+1} with l_j^i
   (a) r_k^i sends l_j^i a message containing the identity of r_k^i's right neighbor
   (b) r_j^i sends his left neighbor a message containing the indentity of l_j^i

In Figure 9 a communication tree is constructed for a subtree containing 5 vertices. In the first iteration, three pairs, (a, b), (c, d), and (e, f) are formed. The roots on the left, a, c, and e become the roots of the subtrees after the mergers, and b, d, and f are removed from the cycle of subtree roots. In the second iteration, c and e pair. Note how the rightmost leaf of c's subtree, d, becomes the root of the subtree resulting from the merge. Now only two subtree roots are left, a and d. Assuming that a has a smaller label than d, the final subtree, communication tree, has b as its root. A quick look at the communication tree reveals that in its inorder traversal, the vertices are visited in the same order that they appear in C'.

6.3 Communication tree Broadcasting Algorithm

Our motivation for building a communication tree CT is to provide a way to quickly gather information from and pass information to all of the vertices in a subtree. We can use the following algorithm to broadcast a message from the root of CT to all of the vertices in CT. The same algorithm can be run in reverse, with each internal node forwarding to his father only one of the messages that he receives from his children.

Algorithm 3 *Communication Tree Broadcasting Algorithm*

> **A message is passed down from the root to all of the nodes in communication tree CT one level at a time. The message is first simultaneously sent from the root of CT to his left child, and then to his right child. When all of his children have received the message, they forward it to their left children, and then to their right children. The process is repeated until the leaves of CT have received the message.**

7 Minimum Spanning Forest Algorithm

We can efficiently implement Sollin's algorithm by using a communication tree to coordinate communication between the vertices in each subtree in E_T. Recall that the principle step in Sollin's algorithm is to choose for each subtree T_i, the edge of smallest weight connecting a vertex $u \in T_i$ with a vertex $v \in T_j$, $i \neq j$. In our implementation, each vertex $u \in T_i$ sends his neighbors in G the label of the root of CT_i. He then chooses the lightest edge connecting him to a vertex in another communication tree CT_j, breaking ties lexicographically as described before. These potential merging edges are then passed up CT_i, which is used as a decision tree, allowing only the lightest of the edges reaching each internal node to progress up the tree (again we may have to break ties lexicographically). The unique lightest edge reaching the root is used to merge the two trees that it connects, T_i and T_j. If no edge reaches the root, then T_i is a minimum spanning tree for a connected component of G, and the vertices in that component may become idle.

After each merger occurs, a new communication tree CT is constructed from the combined cycles C'_i and C'_j, C'_{ij}, and the root of CT informs the vertices in T_{ij} of his identity.

Note that several trees may choose to merge with T_i, with possibly more than one of them wishing to break C'_i at a single vertex u. We will show how to implement directed Euler cycles so that an arbitrary number of cycles can be efficiently merged at u.

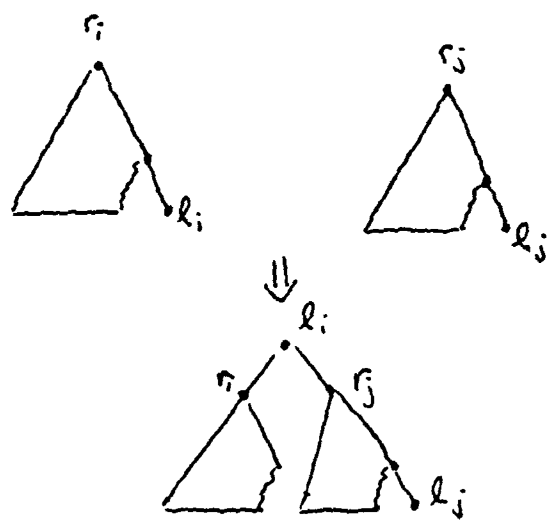


Figure 6: Merging Communication Tree Subtrees

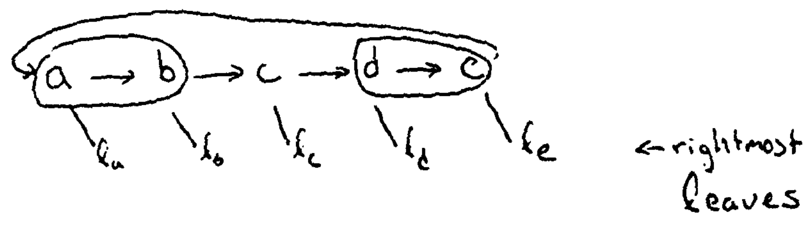


Figure 7: Cycle of Subtree Roots Before Mergers

$$\begin{tikzcd} \ell_a \arrow[r] & c \arrow[r] & \ell_d \arrow[ll, bend right] \end{tikzcd}$$

Figure 8: Cycle of Subtree Roots After mergers

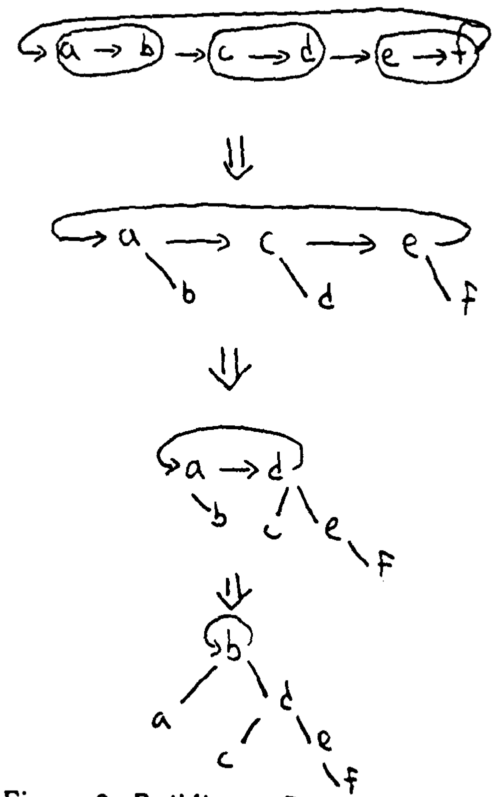


Figure 9: Building a Communication Tree

Algorithm 4 *Minimum Spanning Forest*

```
$E_T = \emptyset$

Each vertex $u_i$ is active and is an isolated subtree, $T_i$.

WHILE there are active vertices DO

    1. Each vertex $u$ sends the label of the root of his communication
       tree to his neighbors in $G$.

    2. Each vertex $u$ determines which edge adjacent to a different tree
       has the smallest weight. In the case of a tie, each edge $u \overset{e}{—} v$
       is given a label $k(e) = (\max(u,v), min(u,v))$ and the edge with
       the smallest label is chosen.

    3. Each vertex $u$ at the leaf of a communication tree passes the edge
       of smallest weight that reaches him to his father in the commu-
       nication tree. The communication tree is used as a decision tree,
       allowing only the smallest edge reaching each internal vertex to
       pass.

    4. The root of each communication tree $CT$ broadcasts the merging
       edge to all of the vertices in $CT$. If no edge has reached the root,
       then the minimum spanning tree of this connected component
       has been found. In this case, the root of $CT$ broadcasts a halt
       instruction.

    5. For each merging edge $u \overset{e}{—} v, u \in T_i, v \in T_j$, merge $C_i'$ and $C_j'$.

    6. For each directed cycle, build a new communication tree.

    7. The new root of each communication tree broadcasts his label.
```

In Figures 10 through 13 we show a sample execution of the MSF algorithm. As Figure 10 shows, the input graph G contains four vertices, each of which has two neighbors. Initially each vertex v_i is an isolated tree T_i, and the set of minimum spanning forest edges, E_T, is empty.

In Figure 11 we show the choices of merging edges made by the v_i in the first iteration. Each vertex chooses the lightest edge connecting him to a vertex in a different subtree. As the figure shows, v_0 chooses e_3, v_1 chooses e_0, v_2 chooses e_3, and v_3 chooses e_1. Note that v_3 must break a tie between edges e_1 and e_2 lexicographically.

At this point each vertex v_i is the sole vertex of a tree T_i, so the edge that v_i picks becomes the merging edge for T_i. In Figure 12 we show the cycle C' constructed from edges e_0, e_1, and e_3. In this example one cycle contains all of the vertices in V. For clarity, we have chosen to label the appearances of the v_i in C' a, b, c, d, e, and f.

In figure 13 we show the communication tree CT constructed from the cycle C'. This tree is exactly that of Figure 9. At the end of the first iteration, b sends all of the vertices in CT his label. Since all of the vertices in V are contained in CT, in the second iteration no vertex finds that he is adjacent to a vertex in any other communication tree and b broadcasts a halt instruction.

8 Implementation of Directed Euler Cycles

Let T_u be a subtree of G with directed Euler cycle C'_u. We want to maintain the cycle C'_u by storing information at the vertices that appear on the C'_u. We also want to quickly merge two cycles, C'_u and C'_v by adding two oppositely direct edges between a vertex $u_i \in V_u$ and $v_j \in V_v$. Thus for each edge $e \in E'_u$ adjacent to u_i, we store $\text{out}_{u_i}(e)$, the edge to traverse out from u_i after e is traversed in to u_i. In order to merge two cycles C'_u and C'_v by adding edges e_{ij} and e_{ji} with endpoints $u_i \in V_u$ and $v_j \in V_v$ we pick an edge $e' \in E_u$ adjacent to u_i and an edge $e'' \in E_v$ adjacent to v_j perform the following operations:

$$\begin{aligned}
\text{out}_{u_i}(e_{ji}) &:= \text{out}_{u_i}(e') \\
\text{out}_{u_i}(e') &:= e_{ij} \\
\text{out}_{v_j}(e_{ij}) &:= \text{out}_{v_j}(e'') \\
\text{out}_{v_j}(e'') &:= e_{ij}
\end{aligned}$$

These operations are illustrated in Figure 14.

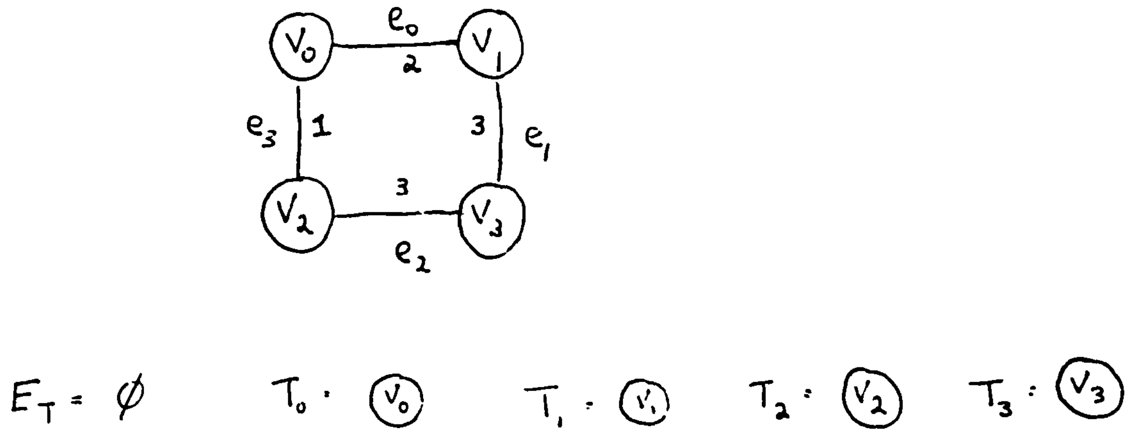


Figure 10: Input Graph G

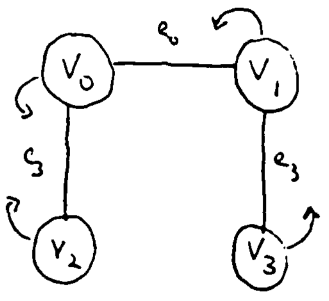


Figure 11: Merging Edges

$$\begin{tikzcd}
V_0 \arrow[r] & V_1 \arrow[r] & V_3 \arrow[r] & V_1 \arrow[r] & V_0 \arrow[r] & V_2 \arrow[lllll, bend right] \\
a & b & c & d & e & f
\end{tikzcd}$$

Figure 12: Directed Euler Cycle

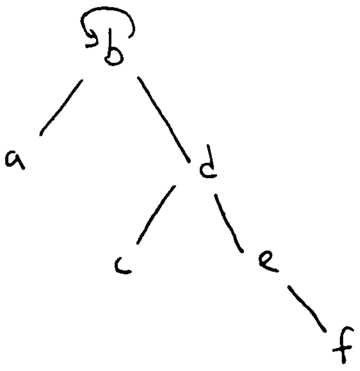


Figure 13: Communication Tree

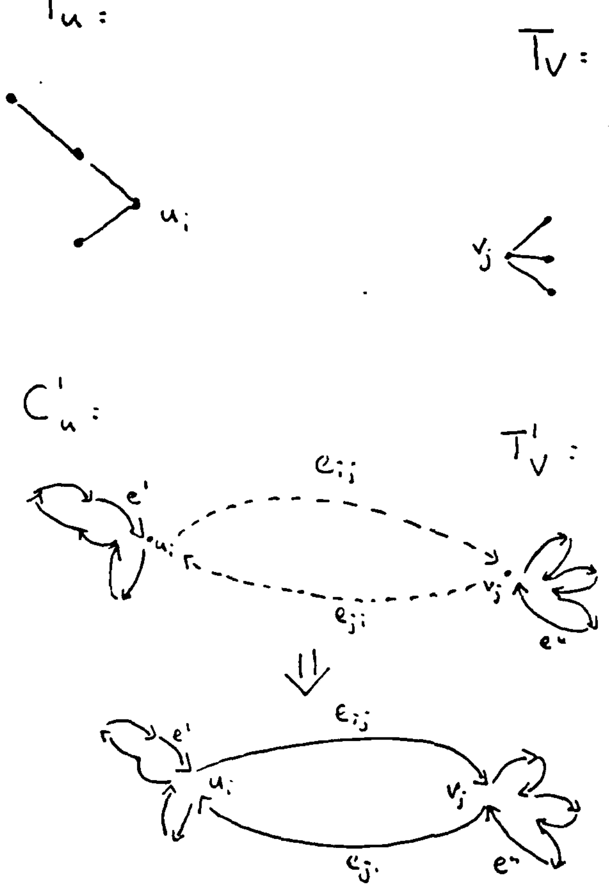


Figure 14: Cycle Merging Operations

9 Pairing Algorithms

Let C_i' be the directed Euler cycle of tree T_i. A vertex v may appear more than once in such a cycle. Suppose vertex v_i appears after vertex v_{i-1} in C_i'. We can give v the label $k(v) = (v_{i-1}, v_i)$. No other vertex can have the same label, because the cycle is Eulerian. For simplicity, we shall speak of each of these uniquely labelled appearances of v as if they we different vertices in C_i. We may use the either of the following algorithms to pair the vertices in C_i'.

9.1 Deterministic Pairing Algorithm

Algorithm 5 *Deterministic Pairing*

```
Let $k(v_i)_j$ be bit $j$ of the binary encoding of $k(v_i)$.

For each active vertex $v_i$, perform the following operations synchronously.

Vertex $v_i$ becomes active.

FOR l := low order bit position of $k(v)$ to high order bit position of
$k(v)$ DO

    IF $k(v_i)_l = 1$ and $k(v_{i+1})_l = 0$
        THEN
            Let $u = v_{i+1}$.
            Send a message to $u$.
    IF $l(v_k)_i = 0$ and $l(v_{k-1})_i$ is 1
        THEN
            Let $u = v_{i-1}$.
            Send a message to $u$.
    IF $v_i$ received a message from a vertex $u$
        THEN
            $v_i$ pairs with $u$ and becomes in active
```

Active vertices repeat the process above for $0 \rightarrow 1$ transitions.

In this algorithm we alternate sending messages sets consisting of messages only to right neighbors in C'_i and only to left neighbors. At no point do we intersperse messages to left and right neighbors.

In Figures 15 and 16 we show the execution of the deterministic pairing algorithm on a cycle of length 6. Recall that the same vertex may appear more than once on a cycle, but that each appearance is given a unique label. In Figure 15 the binary representation of the unique label of each vertex appearance is listed below that appearance.

As Figure 16 shows, two pairs are formed when $1 \rightarrow 0$ transitions are examined. The first such transition occurs in the low order bit position from vertex v_3 (appearance $(3,1)$) to vertex v_1 (appearance $(1,0)$). The second pair is found when the transition between v_0 (appearance $(0,2)$) and v_2 (appearance $(2,0)$) in the next bit position is examined. A third pair is found by examining the $0 \rightarrow 1$ transitions. Note that several $1 \rightarrow 0$ and $0 \rightarrow 1$ transitions are ignored because by the time they are examined, one or both of the vertices involved have already paired.

9.2 Probabilistic Pairing Algorithm

As we will later show, the deterministic pairing algorithm has the disadvantage that it requires $O(\log |V|)$ iterations in which very sparse message sets are generated. While the following probabilistic pairing algorithm is not quaranteed to pair a constant fraction of the vertices in cycle C' we will show that using the probabilistic pairing algorithm the communication tree construction algorithm will, with high probability, take the same order of iterations as when the deterministic algorithm is used. The algorithm assumes that a processor has the capability of making a random choice. Recall that this capability is already required by our probabilistic on-line routing algorithm.

Algorithm 6 *Probabilistic Pairing*

Let v_i appear on cycle C' between vertices v_{i-1} and v_{i+1}.

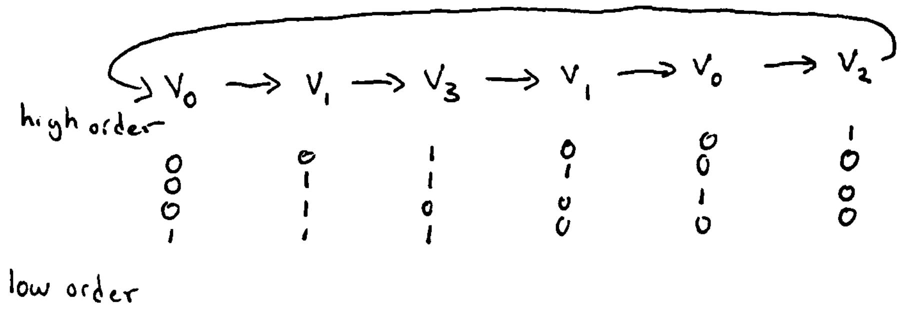


Figure 15: Cycle with Unique Vertex Labels Shown

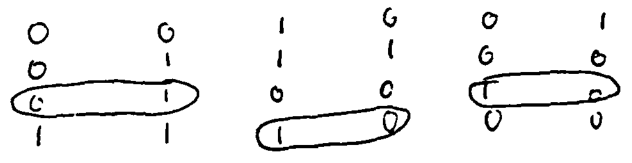


Figure 16: Final Pairings

```
For each vertex $v_i$ perform the following operations synchronously.
Randomly pick $v_{i-1}$ or $v_{i+1}$ to send a message to. Call this vertex $u$.
IF $u = v_{i+1}$
    THEN
        Send a message to $u$.
IF $u = v_{i-1}$
    THEN
        Send a message to $u$.
IF $v_i$ received a message from $u$
    THEN
        $v_i$ pairs with $u$ and becomes inactive
```

10 Analysis

10.1 Deterministic Pairing

Lemma 2 *The deterministic pairing algorithms pairs at least $\frac{2}{3}$ of the vertices in the cycle C'.*

Proof: The label of every vertex v_i, $k(v_i)$ must differ with each of $k(v_{i-1})$ and $k(v_{i+1})$ in at least one bit position l. If the transition from $k(v_{i-1})_l$ to $k(v_i)_l$ is $1 \rightarrow 0$, then either there is a $0 \rightarrow 1$ transition from $k(v_i)_l$ to $k(v_{i+1})_l$ or $k(v_i)_l = k(v_{i+1})_l$. Since we treat $1 \rightarrow 0$ and $0 \rightarrow 1$ transitions in separate iteration loops, every bit difference between the labels of two neighbors is considered individually.

Assume that two neighbors are both unpaired when the algorithm terminates. These two neighbors must differ in at least one bit position. Since this difference was considered in some iteration, they should have paired that iteration. Therefore, no two neighbors can be both unpaired when the algorithm terminates. At the end of the algorithm, then, a vertex can be unpaired only if both of his neighbors paired with other vertices. Thus at least $\frac{2}{3}$ of the vertices in C' are paired when the algorithm terminates. □

Lemma 3 *The deterministic pairing algorithm generates $O(\log |V|)$ message sets, each of which can be delivered in $O(\beta(G))$ delivery cycles.*

Proof: We can encode $k(v)$ in $O(\log |V|)$ bits. The algorithm, therefore, may perform $O(\log |V|)$ iterations.

The algorithm operates on all cycles simultaneously. In each iteration, two separate message sets are generated, one of messages to right neighbors, M_R, and one of messages to left neighbors, M_L. The messages in one of these message sets travel in the direction of the edges in the cycles while the messages in the other travel in the opposite direction.

For every edge $e \in E_T$ between vertices v_i and v_j, directed edges $e_{ij} \in E'$ and $e_{ji} \in E'$ appear once in the cycles of the minimum spanning forest subtrees. Since we are assuming that the capacities of corresponding up and down channels in our fat-tree are equal, traversing a directed edge e_{ij} in reverse is equivalent to traversing e_{ji}, and vice versa. Thus each of the message sets M_R and M_L traverse subsets of the edges in E', and can be delivered in $O(\beta(G))$ delivery cycles. □

Lemma 4 *Using the deterministic pairing algorithm, the communication tree construction algorithm produces trees of height $O(\log |V|)$.*

Proof: In each iteration, $\frac{2}{3}$ of the remaining subtree roots will be paired by the pairing algorithm. In each merger both subtree roots become internal nodes, and the root of the rightmost leaf of the left subtree becomes a subtree root. Therefore, the number of subtree roots is reduced by $\frac{1}{3}$ in each iteration. After $O(log|V|)$ steps, only one root will remain.

As the height of a subtree grows by at most one in each iteration, each resulting communication tree have height $O(log|V|)$. □

10.2 Probabilistic Pairing

We would like to show that with high probability, the communication trees constructed using the probabilistic pairing algorithm will have the same height as those constructed using the deterministic algorithm.

Whenever two subtrees merge into one subtree, we define the right subtree to be the one that merges and the left subtree to be the one that remains in the cycle.

Consider subtree roots $r_i \in V_i$ and $r_{i+1} \in V_{i+1}$ below. When T_i and T_{i+1} merge, T_i is the tree that remains in the cycle.

$\rightarrow r_i \rightarrow r_{i+1} \rightarrow$

Lemma 5 *In any iteration of the CT construction algorithm, a subtree has probability $\frac{3}{4}$ of remaining in the cycle.*

Proof: We are assuming that each vertex chooses to pair with either his left or right neighbor in the cycle with equal probability and independent of the choice of any other vertex. A subtree merges in a given round if the root of that subtree chooses his right neighbor, and his right neighbor chooses him. This probability is computed below.

$$\begin{aligned}\Pr(v_i \text{ chooses } v_{i+1} \text{ and } v_{i+1} \text{ chooses } v_i) &= \Pr(v_i \text{ chooses } v_{i+1}) \Pr(v_{i+1} \text{ chooses } v_i) \\ &= \tfrac{1}{2}\tfrac{1}{2} \\ &= \tfrac{1}{4}\end{aligned}$$

A subtree remains in the cycle whenever it does not merge. This probability is computed below.

$$\begin{aligned}\Pr(T_{i+1} \text{ remains a subtree}) &= 1 - \Pr(v_i \text{ chooses } v_{i+1} \text{ and } v_{i+1} \text{ chooses } v_i) \\ &= \tfrac{3}{4}\end{aligned}$$

□

Lemma 6 *The probability that a subtree T_i will remain in the cycle of subtree roots after m rounds of the communication tree construction algorithm is $\frac{3}{4}$.*

Proof: We are assuming that the choices made in each iteration are independent of the choices made in any other iteration. Let M_i^j be the event that subtree T_i merges in round j. The probability that T_i will merge in round m is simply $\frac{1}{4}$ the probability that T_i did not merge in any of the previous $m-1$ rounds. That is,

$$P(M_i^m) = \frac{3}{4}^{m\ 1}\frac{1}{4}$$

Let M_i be the event that subtree T_i merges in one of the first m iterations. This probability is the sum of a geometric series:

$$\begin{aligned} P(M_i) &= \tfrac{1}{4}\left(\frac{1\ \frac{3}{4}^m}{1\ \frac{3}{4}}\right) \\ &= 1 - \tfrac{3}{4}^m \end{aligned}$$

A subtree remains in the cycle after m iterations when he does not merge in any of the first m iterations. Let R_i be the event that subtree T_i remains in the cycle after m iterations. The probability that a subtree does not merge in any of the first m iterations is expressed below.

$$\begin{aligned} P(R_i) &= 1 - P(M_i) \\ &= \tfrac{3}{4}^m \end{aligned}$$

□

Lemma 7 *The probability that the communication tree construction algorithm will build any communication tree of height greater than greater than* $k \log |V|$ *is* $O(\frac{1}{|V|^{k-1}})$.

Proof: As in the previous lemma, let R_i be the event that subtree T_i remains in the cycle of subtree roots after m iterations. An elementary theorem of probability is that the probability of the union of one or more events is less than or equal to the sum of the individual probabilities of those events. Applying this relation to the R_i, we have

$$\begin{aligned} \Pr(R_1 \cup R_2 \cup \ldots \cup R_{|V|}) &\le P(R_1) + P(R_2) + \ldots + P(R_{|V|}) \\ &\le |V|(\tfrac{3}{4})^m. \end{aligned}$$

For $m = k \log n$ iterations, we have

$$\begin{aligned}
P(R_1 \cup R_2 \cup \ldots \cup R_n) &\leq |V| \frac{3}{4}^{k \log_{4/3} |V|} \\
&\leq |V| \frac{1}{\frac{4}{3}}^{k \log_{4/3} n} \\
&\leq |V| \frac{1}{\frac{4}{3}}^{\log_{4/3} |V|^k} \\
&\leq n \frac{1}{|V|^k} \\
&\leq \frac{1}{|V|^{k-1}}
\end{aligned}$$

□

10.3 *CT* Construction

Lemma 8 *Using the deterministic pairing algorithm, the communication tree construction algorithm generates $O(\log^2 |V|)$ message sets, each of which can be delivered in $O(\beta(G))$ delivery cycles.*

Proof: By lemma 4, the communication tree construction algorithm performs $O(\log |V|)$ iterations in which the deterministic pairing algorithm generates $O(\log |V|)$ messages sets. By lemma 3, each of these message sets can be delivered in $O(\beta(G))$ delivery cycles.

In addition to the message sets generated by the pairing algorithm, the communication tree construction algorithm generates a constant number of messages sets in replacing the left and right subtree roots of each pair with the rightmost leaf of the left subtree. However, by an argument analogous to that of lemma 3, each of these message sets can be delivered in $O(\beta(G))$ delivery cycles. □

Lemma 9 *Using the probabilistic pairing algorithm, the communication tree construction algorithm generates $O(\log |V|)$ message sets, each of which can be delivered in $O(\beta(G))$ delivery cycles.*

Proof: The proof is analogous to the proof of lemma 8. We use the fact that by an argument similar to that of lemma 3, the probabilistic pairing algorithm generates 2 messages sets, each of which can be delivered in $O(\beta(G))$ delivery cycles. □

10.4 Communication Tree Broadcasting

Definition 6 *The projection of an edge $u \overset{e}{—} v$ in a communication tree CT_i where u is the father of v is the path $u \twoheadrightarrow v$ in the directed cycle C_i' when v is a right child, and the path $v \twoheadrightarrow u$ when v is a left child.*

Lemma 10 *The communication tree broadcasting algorithm generates $O(\log|V|)$ messages sets, each of which can be delivered in $\beta(G)$ delivery cycles.*

Proof: Consider the set of father to right child edges at one level of a communication tree CT_i. The projections of these edges are all paths from the fathers to their right children. In an inorder traversal of CT_i we visit the endpoints of these edges from left to right, always visiting a father and his right child consecutively. As these endpoints appear in C_i' in the same order that they appear in the traversal, the projections of these edges are disjoint. As in lemma 3, we can shortcut these disjoint projections in all communication trees without increasing the load factor of $M_{G'}$. A similar argument holds for father to left child edges. Thus we can send the left to right message set and the right to left message set at each level in $O(\beta(G))$ delivery cycles.

CT_i has height $O(\log|V|)$ so that the communication broadcasting algorithm generates $O(\log|V|)$ messages sets, each of which can be delivered in $O(\beta(G))$ delivery cycles. □

10.5 Minimum Spanning Forest

Lemma 11 *Using the deterministic pairing algorithm, the minimum spanning forest algorithm generates $O(\log^3|V|)$ message sets, each of which can be delivered in $O(\beta(G))$ delivery cycles.*

Proof: Sollin's algorithm performs $O(\log|V|)$ iterations [GH]. Below is an analysis of the steps that occur in each iteration of the algorithm.

1. By definition, the message set generated in step 1 can be delivered in $O(\beta(G))$ delivery cycles.

2. No messages are generated in step 2.

3. By lemma 10, step 3 generates $O(\log |V|)$ message sets, each of which can be delivered in $\beta(G)$ delivery cycles.

4. See step 3.

5. See step 1.

6. By lemma 8, step 6 generates $O(\log^2 |V|)$ message sets, each of which can be delivered in $O(\beta(G))$ delivery cycles.

7. See step 3.

□

Lemma 12 *Using the probabilistic pairing algorithm, the minimum spanning forest algorithm generates $O(\log^2 |V|)$ message sets, each of which can be delivered in $O(\beta(G))$ delivery cycles.*

Proof: The proof is analogous to that of lemma 11 in which by lemma 9 step 6 generates $O(\log^2 |V|)$ message sets, each of which can be routed in $O(\beta(G))$ delivery cycles. □

11 Synchronizing Message Sets

We have previously assumed a hardware synchronization mechanism that lets each processor know when the routing of each message set is to start and end, and when each delivery cycle of that message set is to start and end. With fixed length messages, all delivery cycles require the same fixed number of clock cycles. Thus if a processor knows when a message set is to start, he can keep synchronized with each delivery cycle by keeping a local counter.

We would like to remove the need for a hardware message set synchronization mechanism. The difficulty is that the exact number of delivery cycles needed to route a message set is not known until all of the messages in that set have reached their destinations. Furthermore, different processors will finish sending their messages in different delivery cycles. However, by computing off-line a communication tree that contains all of the processors in the fat-tree, we can provide a message set synchronization mechanism with no dedicated hardware other than an increase in each channel capacity of 1.

In Figure 17 we show the cycle from which we will build the synchronization communication tree. This cycle contains the processors in FT in the order that they are visited in an inorder traversal of FT. Figure 18 shows that the the load of each channel due to this cycle is at most 1. Figure 19 shows the process of building the synchronization communication tree. This communication tree is computed once off-line and never changes.

Using the communication tree illustrated in Figure 19, the synchronization algorithm is as follows. To start a message set, processor p_3 broadcasts a start signal down the communication tree. In this case each of the $2log_2|P|$ messages sets generated by the communication tree broadcasting algorithm will require exacty one delivery cycle. When a processor receives a start signal, he begins sending his messages in M in the next delivery cycle. When a processor has finished sending all of his messages, he waits for each of his children in the synchronization communication tree to send him a message confirming that they have finished sending their messages, and then forwards the message to his father in the synchronization communication tree.

The synchronization messages will not reduce the probability of any message in M being successfully transmitted, for although at each concentrator switch the number of messages arriving may have increased by 1, the capacity also increased by 1.

For $load(c) > cap(c)$ we have the following relation:

$$\begin{aligned} \lambda'(c) &= \frac{load(c)+1}{cap(c)+1} \\ &< \frac{load(c)}{cap(c)} \\ &< \lambda(c) \end{aligned}$$

Thus we have actually decreased the load factor of each channel.

These synchronization messages may add an additional $\log|P|+\log\log|P|$ delivery cycles to the number of cycles needed to route M, for example in the case when all processors finish sending their messages in M in the same delivery cycle.

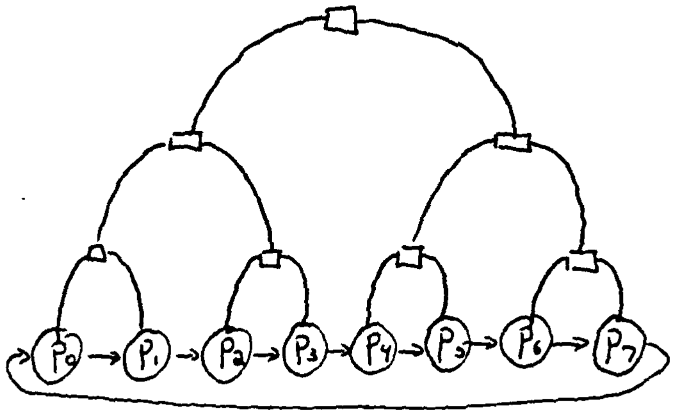


Figure 17: Synchronization Cycle

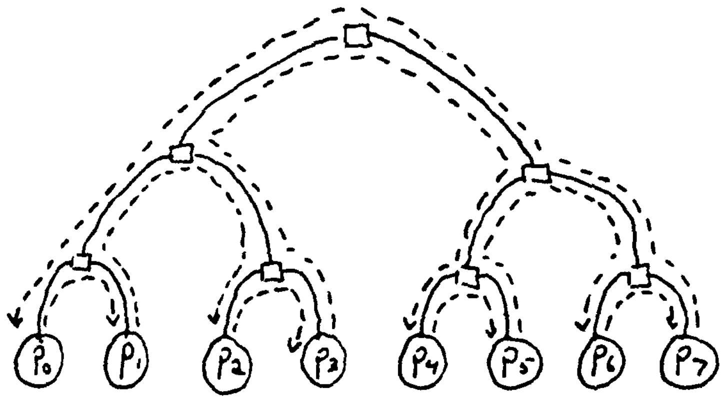


Figure 18: Synchronization Load Factors

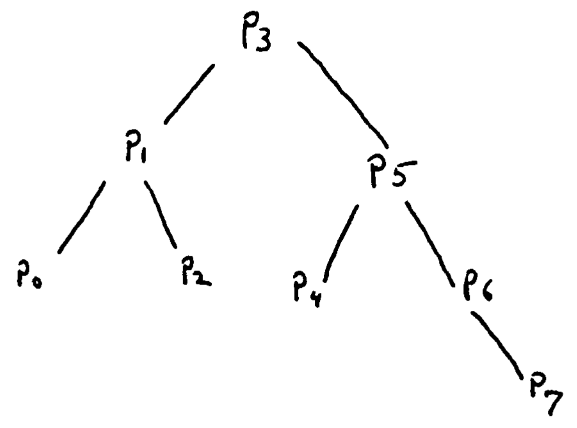


Figure 19: Synchronization Communication Tree

12 Comments

The algorithms described in the previous sections are all SIMD (Single Instruction, Multiple Data) in nature. In each instruction cycle, every processor executes the same instruction. Processors behave differently when they operate on different data. We chose to design our algorithm using the SIMD paradigm only because it is conceptually simpler than the MIMD (Multiple Instruction, Multiple Data) paradigm. We do not mean to imply that parallel algorithms should, in general, use the SIMD paradigm. Similarly, we chose a very simple message passing protocol. The only interaction between a sending processor and a receiving processor is a final acknowledgement. More complicated mechanisms can be realized with essentially the same hardware. For example, instead of passing a message to the receiving processor, the sending processor might send a request to read some portion of the receiving processor's memory. The receiving processor would then reply with that data instead of sending a simple acknowledgement. There may be profound reasons for choosing the SIMD or the MIMD paradigm, or for using some particular message sending protocol, but we have not dealt with these issues in this paper.

In this paper we have examined a technique for keeping communication costs down throughout a parallel algorithm. Our technique is to construct "communication trees" from cycles of processors. If we think of each cycle of processors as a set of processors, then we can imagine using communication trees to implement a variety of basic set operations. Our current algorithms for even such simple operations as computing the union of two sets are very expensive. We must discard the communication tree of each set and build a completely new communication tree. We expect that future research will explore such problems as merging two communication trees directly, and computing the most efficient communication tree for a set of processors.

Finally, the message set routing results of Leiserson and Greenberg [L, LG] show that no matter how large the load factor of a message set, we can, for a given amount of hardware, deliver it in almost optimal time. Thus if a problem takes a long time to run on a fat-tree, then it will take a long time to run on any architecture. These observations lead to the somewhat obvious conclusion that we should examine those problems for which we only need to generate message sets with small load factors.

13 Acknowledgements

I would like to thank Charles Leiserson for guiding me through this research and for insisting that my writing will improve. I would also like to thank Tom Cormen, Ron Greenberg, Alex Ishii, Miller Maley, and Cindy Phillips for their suggestions and encouragement.

14 Bibliography

[AS] B. Awerbuch and Y. Shiloach, New Connectivity and MSF Algorithms for Ultracomputer and PRAM, *Proceedings of the 1983 International Conference on Parallel Processing* (August 1983), 175-179.

[B] J. L. Bentley, A Parallel Algorithm for Constructing Minimum Spanning Trees, *Journal of Algorithms 1* (1980), 51-59.

[E] S. Even, *Graph Algorithms*, Section 1.3, Computer Science Press Inc., Rockville Maryland, 1979.

[H] D. S. Hirschberg, Parallel Algorithms for the Transitive Closure and the Connected Components Problems, *Proceedings of the Eighth Annual ACM Symposium on the Theory of Computing* (1983), 55-57.

[HCS] D. S. Hirschberg, A. K. Chandra, D. V. Sarwate, Computing Connected Components on Parallel Computers, *Communications of the ACM 22*, 8 (August 1979), 461-464.

[KR] S. C. Kwan and W. L. Ruzzo, Adaptive Parallel Algorithms for Finding Minimum Spanning Trees, *Proceedings of the 1984 International Conference on Parallel Processing* (August 1984), 439-443.

[L] C. E. Leiserson, Fat-Trees: Universal Networks for Hardware-Efficient Supercomputing, *1985 International Conference on Parallel Processing*, IEEE, to appear.

[LG] C. E. Leiserson and R. Greenberg, Randomized Routing on Fat-Trees, *Proceedings of the Seventeenth Annual ACM Symposium on the Theory of Computing* (1985).

[SV] Y. Shiloach and U. Vishkin, An $O(\log n)$ Parallel Connectivity Algorithm, *Journal of Algorithms 3* (1982), 57-67.

[HG] S. E. Goodman and S.T. Hedetniemi, *Introduction to the Design and Analysis of Algorithms*, Section 5.5, McGraw-Hill, 1977.

[TV] R. E. Tarjan and U. Vishkin, Finding Biconnected Components and Computing Tree Functions in Logarithmic Parallel Time, *Proceedings of the 25th Annual Symposium on the Foundations of Computer Science* (October 1984), 12-20.